TEHNICA DE PROGRAMARE "GREEDY"

1. Planificarea unor spectacole folosind un număr minim de săli

Considerăm n spectacole $S_1, S_2, \dots, S_n$ pentru care cunoaștem intervalele lor de desfășurare $[s_1, f_1), [s_2, f_2), \dots, [s_n, f_n)$, toate dintr-o singură zi. Să se determine numărul minim de săli necesare astfel încât toate spectacolele să fie programate astfel încât să nu existe suprapuneri în nicio sală. Un spectacol S_j poate fi programat după spectacolul S_i dacă $s_j \geq f_i$.

De exemplu, cele 7 spectacolele de mai jos pot fi planificate folosind minim 3 săli:

spectacole.txt	**programare.txt**

10:00-11:20
09:30-12:10
08:20-09:50
11:30-14:00
12:10-13:10
11:15-13:15
15:00-15:30
	Numar minim de sali: 3
	
	Sala 1:
	  11:15-13:15 Spectacol 6
	
	Sala 2:
	  08:20-09:50 Spectacol 3
	  10:00-11:20 Spectacol 1
	  11:30-14:00 Spectacol 4
	
	Sala 3:
	  09:30-12:10 Spectacol 2
	  12:10-13:10 Spectacol 5
	  15:00-15:30 Spectacol 7

Deoarece dorim să găsim o rezolvare de tip Greedy a acestei probleme, vom încerca să planificăm spectacolele folosind următoarea strategie: vom încerca, pe rând, să planificăm fiecare spectacol într-una dintre sălile deja utilizate (după ultimul spectacol planificat în sala respectivă), iar dacă acest lucru nu este posibil, vom programa spectacolul respectiv într-o sală nouă (i.e., o sală neutilizată până atunci).

Spectacolele pot fi parcurse în mai multe moduri: în ordinea crescătoare a orelor de terminare, în ordinea crescătoare a duratelor sau în ordinea crescătoare a orelor de început. În continuare, vom analiza planificările pe care le vom obține pentru spectacolele din exemplul dat, utilizând una dintre modalități de parcurgere precizate anterior:

a) *în ordinea crescătoare a orelor de terminare*: vom planifica primul spectacol (S3) în Sala 1, al doilea spectacol (S1) se poate planifica tot în sala 1, al treilea spectacol (S2) nu se poate planifica tot în Sala 1, deci va fi planificat într-o sală nouă (Sala 2) ș.a.m.d.

Spectacol	Interval de desfășurare	Sala
S3	08:20 – 09:50	1
S1	10:00 – 11:20	1
S2	09:30 – 12:10	2
S5	12:10 – 13:10	1
S6	11:15 – 13:15	3
S4	11:30 – 14:00	4
S7	15:00 – 15:30	1

Evident, planificarea obținută nu este optimă, deoarece folosește 4 săli în loc de 3!

b) *în ordinea crescătoare a duratelor*: vom planifica primul spectacol (S7) în Sala 1, al doilea spectacol (S5) nu se poate planifica tot în Sala 1 (deoarece nu începe după ultimul spectacol planificat în Sala 1!), deci îl vom planifica într-o sală nouă (Sala 2), al treilea spectacol nu se poate programa niciuna dintre sălile 1 și 2, deci va fi programat într-o sală nouă (Sala 3) ș.a.m.d.

Spectacol	Interval de desfășurare	Durată	Sala
S7	15:00 – 15:30	0:30	1
S5	12:10 – 13:10	1:00	2
S1	10:00 – 11:20	1:20	3
S3	08:20 – 09:50	1:30	4
S6	11:15 – 13:15	2:00	4
S4	11:30 – 14:00	2:30	3
S2	09:30 – 12:10	2:40	5

Evident, nici această planificarea nu este optimă, deoarece folosește 5 săli în loc de 3!

c) *în ordinea crescătoare a orelor de început*: vom planifica primul spectacol (S3) în Sala 1, al doilea spectacol (S2) nu se poate planifica tot în Sala 1, deci îl vom planifica într-o sală nouă (Sala 2), al treilea spectacol (S1) se poate programa tot în Sala 1 ș.a.m.d.

Spectacol	Interval de desfășurare	Sala
S3	08:20 – 09:50	1
S2	09:30 – 12:10	2
S1	10:00 – 11:20	1
S6	11:15 – 13:15	3
S4	11:30 – 14:00	1
S5	12:10 – 13:10	2
S7	15:00 – 15:30	1

Evident, această planificarea este optimă, deoarece folosește tot 3 săli, chiar dacă spectacolele sunt distribuite în săli altfel decât în exemplul dat!

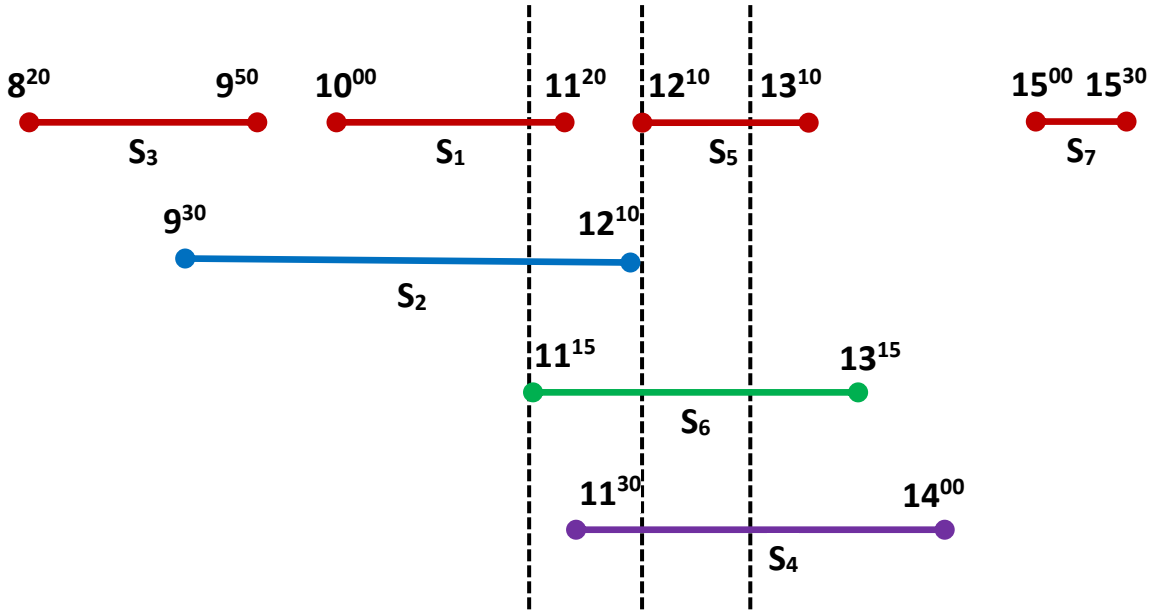
Astfel, analizând exemplele de mai sus, un algoritm Greedy posibil corect, pare a fi următorul:

- sortăm spectacolele în ordinea crescătoare a orelor de început;
- planificăm primul spectacol în prima sală;
- fiecare spectacol rămas îl planificăm fie în prima sală deja utilizată în care acest lucru este posibil (i.e., ora de început a spectacolului curent este mai mare sau egală decât ora de terminare a ultimului spectacol programat în sala respectivă), fie utilizăm o nouă sală pentru el.

În continuare, vom demonstra corectitudinea algoritmului Greedy propus mai sus, folosind următoarele observații:

a) *Algoritmul Greedy nu planifică niciodată două spectacole care se suprapun în aceeași sală*. Argumentați!

b) Definim *adâncimea h a unui șir de intervale de desfășurare ale unor spectacole* ca fiind numărul maxim de spectacole care se suprapun în orice moment posibil. De exemplu, considerând spectacolele din exemplele de mai sus, putem observa faptul că adâncimea șirului intervalelor lor de desfășurare este $h = 3$:

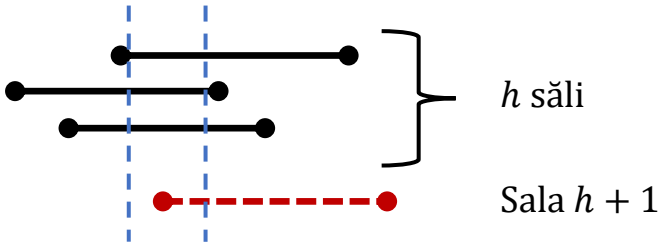


Se observă foarte ușor faptul că *orice planificare corectă a unor spectacole va utiliza un număr de săli cel puțin egal cu adâncimea șirului intervalelor lor de desfășurare*!

c) Pentru un șir de intervale de desfășurare ale unor spectacole având adâncimea h, planificarea realizată de algoritmul Greedy va folosi cel mult h săli.

Demonstrație: Presupunem faptul că algoritmul Greedy ar folosi cel puțin $h + 1$ săli pentru a planifica un șir de spectacole având adâncimea intervalelor lor de

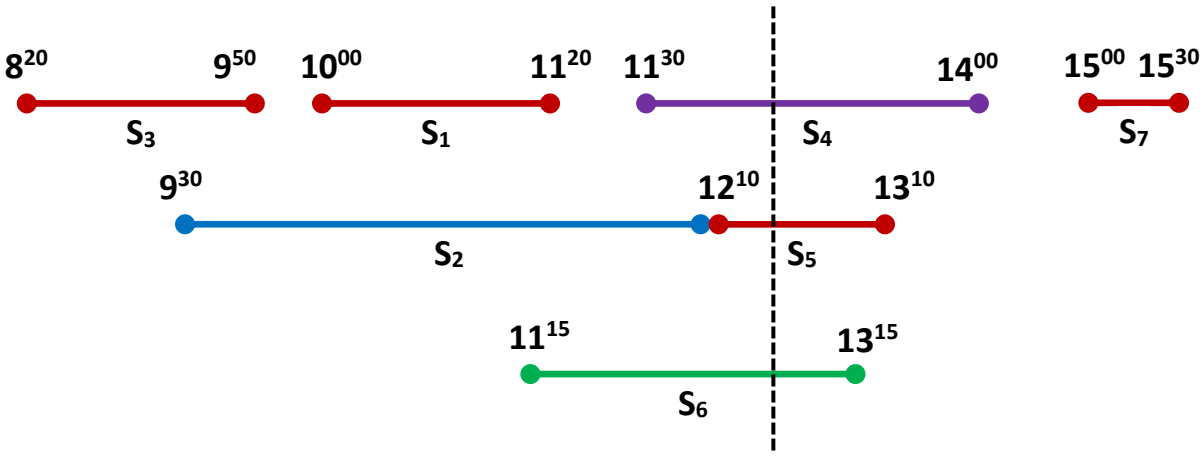
desfășurare egală cu h. Acest lucru s-ar putea întâmpla doar dacă algoritmul Greedy ar fi utilizat deja h săli pentru a planifica spectacolele anterioare spectacolului curent, iar spectacolul curent nu ar putea fi planificat în niciuna dintre ele:



Acest lucru ar însemna faptul că spectacolul curent începe strict înaintea minimului dintre orele de terminare ale spectacolelor deja planificate, deoarece, în caz contrar, spectacolul curent ar fi fost programat într-una dintre sălile deja utilizate. Totodată, știm faptul că spectacolul curent începe după maximul dintre orele de început ale spectacolelor deja planificate, deoarece acestea sunt ordonate crescător după ora de început. Așadar, există un interval de timp (delimitat în figura de mai sus de cele două linii albastre întrerupte) în care spectacolul curent se suprapune cu câte un spectacol din fiecare dintre cele h săli deja utilizate, deci adâncimea șirului de intervale de desfășurare ale spectacolelor respective ar fi $h + 1$. Acest lucru reprezintă o contradicție cu faptul că adâncimea șirului intervalelor de desfășurare ale spectacolelor date este h, deci presupunerea făcută este falsă și, în consecință, algoritmul Greedy va utiliza cel mult h săli pentru a planifica spectacolele respective.

Din observațiile b) și c) rezultă faptul că algoritmul Greedy este optim, deoarece va utiliza exact h săli pentru a planifica spectacole având adâncimea șirului intervalelor de desfășurare egală cu h.

Pentru spectacolele din exemplul utilizat, având adâncimea șirului intervalelor de desfășurare $h = 3$, o planificare optimă (i.e., care folosește 3 săli) este următoarea:



În limbajul Python se poate implementa ușor acest algoritm, păstrând sălile într-o listă, iar fiecare sală va fi tot o listă conținând spectacolele planificate în ea. Totuși, această variantă de implementare ar avea complexitatea maximă $\mathcal{O}(n^2)$, deoarece ora de început a fiecăruia dintre cele n spectacole date ar trebui să fie comparată cu ora de terminare a ultimului spectacol planificat în fiecare dintre cele maxim n săli.

O implementare cu complexitatea optimă $\mathcal{O}(n\log_2 n)$ necesită utilizarea unei structuri de date numită *coadă cu priorități* (*priority queue*). Într-o astfel de structură de date, fiecărei valori îi este asociat un număr întreg reprezentând prioritatea sa, iar operațiile specifice unei cozi se realizează în funcție de prioritățile elementelor, ci nu în funcție de ordinea în care ele au fost inserate. Astfel, operația de inserare a unui nou element și operația de extragere a elementului cu prioritate minimă/maximă (depinde de tipul cozii cu priorități, respectiv *min-priority queue* sau *max-priority queue*) vor avea, de obicei, complexitatea $\mathcal{O}(\log_2 n)$.

În limbajul Python, clasa `PriorityQueue` implementează o coadă cu priorități în care un element trebuie să fie un tuplu de forma $(prioritate, valoare)$. Principalele metode ale acestei clase sunt:

- `get()`: furnizează elementul din coadă care are prioritatea minimă sau, dacă există mai multe elemente cu prioritate minimă, pe primul inserat;
- `put(element)`: inserează în coadă un element de forma indicată mai sus;
- `empty()`: returnează `True` în cazul în care coada nu mai conține niciun element sau `False` în caz contrar;
- `qsize()`: returnează numărul de elemente din coadă.

Așa cum deja am menționat, metodele `get` și `put` au complexitatea $\mathcal{O}(\log_2 n)$, iar metodele `empty` și `qsize` au complexitatea $\mathcal{O}(1)$.

În implementarea algoritmului Greedy prezentat, vom folosi o coadă cu priorități pentru a memora sălile utilizate, prioritatea unei săli fiind dată de ora de terminare a ultimului spectacol planificat în ea, iar spectacolele planificate într-o sală vor fi păstrate folosind o listă. Astfel, vom extrage sala cu timpul minim de terminare al ultimului spectacol planificat în ea și vom verifica dacă spectacolul curent poate fi planificat în sala respectivă sau nu (dacă spectacolul curent nu poate fi planificat în această sală, atunci el nu poate fi planificat în nicio altă sală!). În caz afirmativ, vom planifica spectacolul curent în sala respectivă și îi vom actualiza prioritatea la ora de terminare a spectacolului adăugat, altfel vom insera în coadă o sală nouă în care vom planifica spectacolul curent și prioritatea sălii va fi egală cu ora de terminare a spectacolului respectiv.

Considerând faptul că fișierele de intrare și ieșire sunt de forma prezentată în primul exemplu, o implementare în limbajul Python a algoritmului Greedy este următoarea:

```
import queue

# functie folosita pentru sortarea crescătoare a spectacolelor
# în raport de ora de început (cheia)
def cheieOraÎnceput(sp):
    return sp[1]

# citim datele de intrare din fișierul text "spectacole.txt"
fin = open("spectacole.txt")
# lsp = lista spectacolelor, fiecare spectacol fiind memorat
# sub forma unui tuplu (ID, ora de început, ora de sfârșit)
lsp = []
```

```
crt = 1
for linie in fin:
    aux = linie.split("-")
    # aux[0] = ora de început a spectacolului curent
    # aux[1] = ora de sfârșit a spectacolului curent
    lsp.append((crt, aux[0].strip(), aux[1].strip()))
    crt = crt + 1
fin.close()
# sortăm spectacolele crescător după orelor de început
lsp.sort(key=cheieOraÎnceput)

# sălile vor fi stocate într-o coadă cu priorități în care
# prioritatea unei săli este dată de ora de terminare a
# ultimului spectacol planificat în sala respectivă, iar
# spectacolele planificate în ea vor fi păstrate într-o listă
sali = queue.PriorityQueue()

# planificăm primul spectacol în prima sală
sali.put((lsp[0][2], list((lsp[0],))))

# parcurgem restul spectacolelor
for k in range(1, len(lsp)):
    # extragem sala cu ora minimă de terminare a ultimului
    # spectacol planificat în ea
    min_timp_final = sali.get()

    # dacă spectacolul curent lsp[k] poate fi planificat în
    # sala extrasă, atunci îl adăugăm în lista spectacolelor
    # planificate în ea și reintroducem sala în coada cu
    # priorități, dar cu prioritatea actualizată la ora de
    # terminare a spectacolului adăugat
    if lsp[k][1] >= min_timp_final[0]:
        min_timp_final[1].append(lsp[k])
        sali.put((lsp[k][2], min_timp_final[1]))
    # dacă spectacolul curent lsp[k] nu poate fi planificat în
    # sala extrasă, atunci reintroducem sala extrasă în coada
    # cu priorități fără a-i modifica prioritatea și adăugăm
    # o sală nouă în care planificăm spectacolul curent
    else:
        sali.put(min_timp_final)
        sali.put((lsp[k][2], list((lsp[k],))))

# scriem datele de ieșire în fișierul text "programare.txt"
fout = open("programare.txt", "w")
fout.write("Numar minim de sali: " + str(sali.qsize()) + "\n")

scrt = 1
while not sali.empty():
    sala = sali.get()
    fout.write("\nSala " + str(scrt) + ":\n")
```

```
    for sp in sala[1]:
        fout.write("\t"+sp[1]+"-"+sp[2]+" Spectacol "+
                   str(sp[0]) + "\n")
    scrt += 1

fout.close()
```

Observaţi faptul că în fişierul de ieşire sălile vor fi scrise în ordinea priorităţilor, ci nu în ordinea în care au fost inserate!

2. Problema rucsacului (varianta continuă/fracţionară)

Considerăm un rucsac având capacitatea maximă G şi n obiecte $O_1, O_2, \dots, O_n$ pentru care cunoaştem greutăţile lor $g_1, g_2, \dots, g_n$ şi câştigurile $c_1, c_2, \dots, c_n$ obţinute prin încărcarea lor completă în rucsac. Ştiind faptul că orice obiect poate fi încărcat şi fracţionat (doar o parte din el), să se determine o modalitate de încărcare a rucsacului astfel încât câştigul total obţinut să fie maxim. Dacă un obiect este încărcat fracţionat, atunci vom obţine un câştig direct proporţional cu fracţiunea încărcată din el (de exemplu, dacă vom încărca doar o treime dintr-un obiect, atunci vom obţine un câştig egal cu o treime din câştigul integral asociat obiectului respectiv).

În afara variantei continue/fracţionare a problemei rucsacului, mai există şi varianta discretă a sa, în care un obiecte poate fi încărcat doar complet. Varianta respectivă nu se poate rezolva corect utilizând metoda Greedy, ci există alte metode de rezolvare, pe care le vom prezenta în cursul dedicat metodei programării dinamice.

Se observă foarte uşor faptul că varianta fracţionară a problemei rucsacului are întotdeauna soluţie (evident, dacă $G > 0$ şi $n \geq 1$), chiar şi în cazul în care cel mai mic obiect are o greutate strict mai mare decât capacitatea G a rucsacului (deoarece putem să încărcăm şi fracţiuni dintr-un obiect), în timp ce varianta discretă nu ar avea soluţie în acest caz.

Deoarece dorim să găsim o rezolvare de tip Greedy pentru varianta fracţionară a problemei rucsacului, vom încerca să încărcăm obiectele în rucsac folosind unul dintre următoarele criterii:

a) în ordinea descrescătoare a câştigurilor integrale (cele mai valoroase obiecte ar fi primele încărcate);
b) în ordinea crescătoare a greutăţilor (cele mai mici obiecte ar fi primele încărcate, deci am încărca un număr mare de obiecte în rucsac);
c) în ordinea descrescătoare a greutăţilor.

Analizând cele 3 criterii propuse mai sus, putem găsi uşor contraexemple care să dovedească faptul că nu vom obţine o soluţii optime. De exemplu, criteriul c) ar putea fi corect doar presupunând faptul că, întotdeauna, un obiect cu greutate mare are asociat şi un câştig mare, ceea ce, evident, nu este adevărat! În cazul criteriului a), considerând $G =$

10 kg şi 3 obiecte având câştigurile $(100, 90, 80)$ RON şi greutăţile $(10, 5, 5)$ kg, vom încărca în rucsac primul obiect (deoarece are cel mai mare câştig integral) şi nu vom mai putea încărca niciun alt obiect, deci câştigul obţinut va fi de 100 RON. Totuşi, câştigul maxim de 170 RON se obţine încărcând în rucsac ultimele două obiecte! În mod asemănător (de exemplu, modificând câştigurilor obiectelor anterior menţionate în $(100, 9, 8)$ RON) se poate găsi un contraexemplu care să arate faptul că nici criteriul b) nu permite obţinerea unei soluţii optime în orice caz.

Se poate observa faptul că primele două criterii nu conduc întotdeauna la soluţia optimă deoarece ele iau în considerare fie doar câştigurile obiectelor, fie doar greutăţile lor, deci criteriul corect de selecţie trebuie să le ia în considerare pe ambele. Intuitiv, pentru a obţine un câştig maxim, trebuie să încărcăm mai întâi în rucsac obiectele care sunt cele mai "eficiente", adică au un câştig mare şi o greutate mică. Această "eficienţă" se poate cuantifica prin intermediul *câştigului unitar* al unui obiect, adică prin raportul $u_i = c_i/g_i$.

Algoritmul Greedy pentru rezolvarea variantei fracţionare a problemei rucsacului este următorul:

- sortăm obiectele în ordinea descrescătoare a câştigurilor unitare;
- pentru fiecare obiect testăm dacă încape integral în spaţiul liber din rucsac, iar în caz afirmativ îl încărcăm complet în rucsac, altfel calculăm fracţiunea din el pe care trebuie să o încărcăm astfel încât să umplem complet rucsacul (după încărcarea oricărui obiect, actualizăm spaţiul liber din rucsac şi câştigul total);
- algoritmul se termină fie când am încărcat toate obiectele în rucsac (în cazul în care $g_1 + g_2 + \cdots + g_n \leq G$), fie când nu mai există spaţiu liber în rucsac.

De exemplu, să considerăm un rucsac în care putem să încărcăm maxim $G = 53$ kg şi $n = 7$ obiecte, având greutăţile $g = (10, 5, 18, 20, 8, 40, 20)$ kg şi câştigurile integrale $c = (30, 40, 36, 10, 16, 30, 20)$ RON. Câştigurile unitare ale celor 7 obiecte sunt $u = \left(\frac{30}{10}, \frac{40}{5}, \frac{36}{18}, \frac{10}{20}, \frac{16}{8}, \frac{30}{40}, \frac{20}{20}\right) = (3, 8, 2, 0.5, 2, 0.75, 1)$ RON/kg, deci sortând descrescător obiectele în funcţie de câştigul unitar vom obţine următoarea ordine a lor: $O_2, O_1, O_3, O_5, O_7, O_6, O_4$. Prin aplicarea algoritmului Greedy prezentat anterior asupra acestor date de intrare, vom obţine următoarele rezultate:

Obiectul curent	**Fracţiunea încărcată din obiectul curent**	**Spaţiul liber în rucsac**	**Câştigul total**
—	—	53	0
O_2: $c_2 = 40, g_2 = 5 \leq 53$	1	53 – 5 = 48	0 + 40 = 40
O_1: $c_1 = 30, g_1 = 10 \leq 48$	1	48 – 10 = 38	40 + 30 = 70
O_3: $c_3 = 36, g_3 = 18 \leq 38$	1	38 – 18 = 20	70 + 36 = 106
O_5: $c_5 = 16, g_5 = 8 \leq 20$	1	20 – 8 = 12	106 + 16 = 122
O_7: $c_7 = 20, g_7 = 20 > 12$	12/20 = 0.6	0	122 + 0.6*20 = 134

În concluzie, pentru a obține un câștig maxim de 134 RON, trebuie să încărcăm integral în rucsac obiectele O_2, O_1, O_3, O_5 și o fracțiune de $0.6 = \frac{3}{5}$ din obiectul O_7.

Înainte de a demonstra corectitudinea algoritmului prezentat, vom face următoarele observații:

- vom considera obiectele $O_1, O_2, \dots, O_n$ ca fiind sortate strict descrescător în funcție de câștigurile lor unitare, respectiv $\frac{c_1}{g_1} > \frac{c_2}{g_2} > \cdots > \frac{c_n}{g_n}$ (acest lucru se obține grupând obiectele care au același câștig unitar, ceea ce nu afectează corectitudinea algoritmului Greedy deoarece obiectele pot fi fracționate și vor fi încărcate în rucsac în aceeași ordine);
- o soluție a problemei va fi reprezentată sub forma unui tuplu $X = (x_1, x_2, \dots, x_n)$, unde $x_i \in [0,1]$ reprezintă fracțiunea selectată din obiectul O_i;
- în toate formulele vom considera implicit indicii ca fiind cuprinși între 1 și n;
- câștigul asociat unei soluții a problemei de forma $X = (x_1, x_2, \dots, x_n)$ îl vom nota cu $C(X) = \sum c_i x_i$;
- dacă $g_1 + g_2 + \cdots + g_n \leq G$, atunci soluția vom obține soluția banală $X = (1, \dots, 1)$, care este evident optimă, deci vom considera faptul că $g_1 + g_2 + \cdots + g_n > G$, precum și faptul că în orice soluție optimă rucsacul va fi umplut complet (adică $\sum g_i x_i = G$).

Fie $X = (x_1, x_2, \dots, x_n)$ soluția furnizată de algoritmul Greedy prezentat, deci rucsacul va fi umplut complet (i.e., $\sum g_k x_k = G$). Presupunem că soluția X nu este optimă, deci există o altă soluție optimă $Y = (y_1, y_2, \dots, y_n)$, diferită de soluția X, obținută folosind un alt algoritm (nu neapărat unul de tip Greedy). Deoarece Y este o soluție optimă, obținem imediat următoarele două relații: $\sum g_i y_i = G$ și câștigul $C(Y) = \sum c_i y_i$ este maxim.

Deoarece $X \neq Y$, rezultă că există un cel mai mic indice i pentru care $x_i \neq y_i$. Mai mult, datorită faptului că algoritmul Greedy încarcă din fiecare obiect cea mai mare fracțiune posibilă, rezultă că $x_i > y_i$. Totuși, soluția Y este optimă, deci $C(Y) > C(X)$, ceea ce înseamnă că $\exists j > i$ astfel încât $x_j < y_j$ (altfel, respectiv dacă $x_k \geq y_k$ pentru orice indice $k > i$, am avea $C(X) > C(Y)$, ceea ce ar contrazice optimalitatea soluției Y).

Considerăm acum soluția $Z = (z_1, z_2, \dots, z_i, \dots, z_j, \dots, z_n)$, obținută din soluția Y prin mutarea unei fracțiuni ε de la obiectul j la obiectul i (i.e., vom încărca în rucsac mai puțin din obiectul j și mai mult din obiectul i, deoarece obiectul i are un câștig unitar strict mai mare decât cel al obiectului j) astfel încât rucsacul să rămână umplut complet:

- $z_k = y_k$ pentru orice $k \neq i, j$;
- $z_j = y_j - \varepsilon$;
- $z_i = y_i + \varepsilon \frac{g_j}{g_i}$.

Valoarea fracțiunii z_i se poate calcula astfel: dacă din obiectul j vom încărca o fracțiune mai mică cu ε (i.e., fracțiunea $y_j - \varepsilon$), atunci greutatea totală a obiectelor din rucsac va scădea cu $\varepsilon \cdot g_j$ unități de masă (e.g., kilograme), deci pentru a păstra rucsacul încărcat complet trebuie să luăm mai mult cu $\varepsilon \cdot g_j$ unități de masă din obiectul i, ceea ce înseamnă

o fracţiune de $\frac{\varepsilon \cdot g_j}{g_i} = \varepsilon \frac{g_j}{g_i}$ din masa sa. Evident, valoarea fracţiunii ε trebuie să fie aleasă în mod convenabil, respectiv $0 < \varepsilon \leq y_j$ şi astfel încât $z_i \leq 1$!

În continuare, calculăm câştigul asociat soluţiei Z:

$$C(Z) = \sum z_k c_k = \sum_{k \neq i,j} y_k c_k + z_i c_i + z_j c_j = \sum_{k \neq i,j} y_k c_k + \left(y_i + \varepsilon \frac{g_j}{g_i}\right) c_i + (y_j - \varepsilon) c_j$$

$$= \underbrace{\sum_{k \neq i,j} y_k c_k + y_i c_i + y_j c_j}_{C(Y)} - \varepsilon c_j + \varepsilon \frac{g_j}{g_i} c_i = C(Y) + \underbrace{\varepsilon \left(\frac{c_i g_j - c_j g_i}{g_i}\right)}_{>0}$$

Expresia $\varepsilon \left(\frac{c_i g_j - c_j g_i}{g_i}\right)$ este strict pozitivă deoarece $\frac{c_i}{g_i} > \frac{c_j}{g_j} \Leftrightarrow c_i g_j - c_j g_i > 0$, iar ε şi g_i sunt evident strict pozitive, deci rezultă că $C(Z) > C(Y)$, ceea ce contrazice optimalitatea soluţiei Y, aşadar presupunerea că ar exista o soluţie optimă Y diferită de soluţia X furnizată de algoritmul Greedy este falsă.

În continuare, vom prezenta implementarea în limbajul Python a algoritmului Greedy pentru rezolvarea variantei fracţionare a problemei rucsacului:

```
# functie folosita pentru sortarea descrescătoare a obiectelor
# în raport de câștigul unitar (cheia)
def cheieCâștigUnitar(ob):
    return ob[2] / ob[1]
# citim datele de intrare din fișierul text "rucsac.in"
fin = open("rucsac.in")
# de pe prima linie citim capacitatea G a rucsacului
G = float(fin.readline())
# fiecare dintre următoarele linii conține
# greutatea și câștigul unui obiect
obiecte = []
crt = 1
for linie in fin:
    aux = linie.split()
    # un obiect este un tuplu (ID, greutate, câștig)
    obiecte.append((crt, float(aux[0]), float(aux[1])))
    crt += 1
fin.close()

# sortăm obiectele descrescător în funcție de câștigul unitar
obiecte.sort(key=cheieCâștigUnitar, reverse=True)
# n reprezintă numărul de obiecte
n = len(obiecte)
# solutie este o listă care va conține fracțiunile încărcate
# din fiecare obiect
soluție = [0] * n
# inițial, spațiul liber din rucsac este chiar G
spațiu_liber_rucsac = G
```

```
# considerăm, pe rând, fiecare obiect
for i in range(n):
    # dacă obiectul curent încape complet în spațiul liber
    # din rucsac, atunci îl încărcăm complet
    if obiecte[i][1] <= spațiu_liber_rucsac:
        spațiu_liber_rucsac -= obiecte[i][1]
        soluție[i] = 1
    else:
        # dacă obiectul curent nu încape complet în spațiul liber
        # din rucsac, atunci calculăm fracțiunea din el necesară
        # pentru a încărca complet rucsacul și algoritmul se termină
        soluție[i] = spațiu_liber_rucsac / obiecte[i][1]
        break

# calculăm câștigul maxim
câștig = sum([soluție[i] * obiecte[i][2] for i in range(n)])

# scriem datele de ieșire în fișierul text "rucsac.out"
fout = open("rucsac.out", "w")
fout.write("Castig maxim: " + str(câștig) + "\n")
fout.write("\nObiectele incarcate:\n")
i = 0
while i < n and soluție[i] != 0:
    # trunchiem procentul încărcat din obiectul curent
    # la două zecimale
    procent = format(soluție[i]*100, '.2f')
    fout.write("Obiect "+str(obiecte[i][0])+": "+procent+"%\n")
    i = i + 1
fout.close()
```

Pentru exemplul de mai sus, fișierele text de intrare și de ieșire sunt următoarele:

rucsac.in	**rucsac.out**
53 10 30 5 40 18 36 20 10 8 16 40 30 20 20	Castig maxim: 134.0 Obiectele incarcate: Obiect 2: 100.00% Obiect 1: 100.00% Obiect 3: 100.00% Obiect 5: 100.00% Obiect 7: 60.00%

Citirea datelor de intrare are complexitatea $\mathcal{O}(n)$, sortarea are complexitatea $\mathcal{O}(n \log_2 n)$, selectarea și încărcarea obiectelor în rucsac are cel mult complexitatea $\mathcal{O}(n)$, iar afișarea câștigului maxim obținut $\mathcal{O}(1)$, deci complexitatea algoritmului este $\mathcal{O}(n \log_2 n)$.